

ON THE POSSIBILITY OF SIMPLE PARALLEL COMPUTING OF VORONOI DIAGRAMS AND DELAUNAY GRAPHS

DANIEL REEM

ABSTRACT. Although many algorithms for computing Euclidean Voronoi diagrams of point sites have been published, most of them are sequential in nature and hence cast inherent difficulties on the possibility to compute the diagrams in parallel. This paper presents a new algorithm which enables the (combinatorial) computation of the diagram. The algorithm is significantly different from previous ones and some of the involved concepts in it are in the spirit of convex analysis and optics. Parallel implementation is naturally supported since each Voronoi cell can be computed independently of the other cells. A new combinatorial structure for representing the cells is described along the way and the computation of the induced Delaunay graph follows as a simple consequence.

Date: December 4, 2012.

2010 *Mathematics Subject Classification.* 68U05, 68W10, 65D18.

Key words and phrases. Algorithm, cone, conic beam, Delaunay graph, parallel computing, ray, subface, vertex, Voronoi diagram.

IMPA - Instituto Nacional de Matemática Pura e Aplicada, Estrada Dona Castorina 110, Jardim Botânico, CEP 22460-320, Rio de Janeiro, RJ, Brazil.

E-mail: dream@impa.br .

1. INTRODUCTION

1.1. Background. In its simplest and widely spread form, the Voronoi diagram (The Voronoi tessellation, Dirichlet tessellations) is a certain decomposition of the Euclidean plane (or a region X in the plane) into cells induced by a collection of distinct points P_1, \dots, P_n (called the sites or the generators) and the Euclidean distance. More precisely, the Voronoi cell R_k associated with the site P_k is the set of all the points in X whose distance to P_k is not greater than their distance to the other sites P_j . This definition can be easily generalized to other setting, e.g., \mathbb{R}^m , sites having more general form, various distance functions, etc.

Because they appear in many fields in science and technology and have numerous applications, these diagrams have attracted a lot of attention during the last decades [6, 7, 39, 27], especially in the case of 2-dimensional Euclidean Voronoi diagrams of point sites. In particular, many algorithms for computing these diagrams in the above mentioned setting have been published. Among them we mention the naive method [39, pp. 230-233], the divide-and-conquer method [3], [39, pp. 251-257], [46], the incremental method [29], [30], [38], [39, pp. 242-251], the plane sweep method [26], [39, pp. 257-264], methods based on geometric transforms such as convex hulls [8, 13, 14, 15] or Delaunay tessellations [31], [39, pp. 275-80], methods based on lower envelopes [45], [47, p. 241] and methods for very specific configurations [2].

However, most of these algorithms are sequential in nature and they cannot compute each of the cells independently of the other ones. Instead, they consider the diagram as a combinatorial structure and compute it as a whole in a sequential way. This fact casts inherent difficulties on any attempt to implement these algorithm in a parallel computing environment. Despite this, the possibility to compute the diagrams in parallel has been investigated in a few places, e.g., in [1, 17, 18, 25, 28, 33, 36, 41]. In these works the idea is to somehow share the work between the many processing units (under certain assumptions on the computational model), but because of the sequential nature of the involved algorithms, these processing units must cooperate between themselves and cannot work independently. The implementation of many of these algorithms is not so simple. In addition, a common assumption in the above works is that there are many processing units, e.g., $O(n)$, where n is the number of sites. This obviously cast difficulties on a practical implementation when the number of sites is large.

In addition to the above works, several other works aiming at the parallel computing of geometric structures related to 2D Euclidean Voronoi diagrams (such as Delaunay triangulation) have been published, e.g., [4, 5, 11, 20, 37, 43, 44, 50, 51, 52], but either they are based on the previous works (or vice versa), or they use somewhat similar techniques or similar assumptions on the computational models and the number of processing units.

The motivation for developing parallel-in-nature algorithms for computational tasks stems from several reasons. One important reason is the ability to compute in a fast manner much larger inputs than computed today, in numerous fields, or to perform in a fast way computations which require many iterations, such as centroidal Voronoi

diagrams (CVD) [23, 24]. Another reason is that in recent years most of the computing devices (standard computers, cell phones, graphic processors, etc.) arrive with several (sometimes with hundreds or even thousands) processing units (cores) which are just waiting for being used. Large networks of such computing devices can be also used for parallel computing tasks.

Taking into account all of the above, it is natural to ask whether there exists an algorithm which can compute each of the Voronoi cells independently of the other ones, and hence can provide a simple way to compute the Voronoi diagram in parallel. To the best of our knowledge, only one such an algorithm has been discussed in the literature, namely the naive one which computes each of the cells by intersecting corresponding halfspaces [39, pp. 230-233]. This algorithm is simple, but it is relatively slow: $O(n^2 \log(n))$ for n sites in the worst case. As claimed in [10], on the average (under the assumption of uniform distribution) its time complexity should behave as $O(n)$. We have not seen any implementation which confirms this, and, as a matter of fact, it seems that in general this algorithm is not used frequently. However, a variation of this naive algorithm for the Delaunay triangulation case has appeared very recently in [16], but a careful verification of the data given there shows that when the sites are generated according to the uniform distribution, the implementation behaves in a way which is worse than $O(n)$.

Recently [42], a new algorithm which allows the approximate computation of Voronoi diagrams in a general setting (general sites, general norms, general dimension) was introduced. This algorithm is based on the possibility to represent each cell as a union of rays (line segments), and it approximates the cells by considering a plurality of approximating rays. See Figures 1-2 for an illustration. This algorithm allows the computation of each cell independently of the other ones. However, although in principle the algorithm may compute the combinatorial structure of the cell (e.g., its vertices), this is done in a non-immediate and non-efficient way, since for doing this it needs to somehow detect the corresponding combinatorial components and for achieving this task many rays should be considered and the information obtained from them should be correctly analyzed. What is not clear in advance is to which direction to shoot a ray such that it will hit a vertex exactly. It is therefore natural to ask whether this algorithm can somehow be modified in a such a way that it will allow a simple and efficient computation of the combinatorial structure.

1.2. Contribution of this work. This paper presents and analyzes an algorithm which enables the combinatorial computation of Euclidean Voronoi diagrams of point sites, where each cell is computed independently of the other ones. Parallel implementation is therefore naturally supported. The algorithm is significantly different from previous ones and some of the involved concepts are in the spirit of convex analysis and optics. A schematic description of the algorithm is presented for any dimension, but the details are given only for the 2-dimensional case. In contrast with many algorithms, the sites can be in any position (no “general position” assumption to avoid degenerate cases is made). A new combinatorial structure for representing the cells is described along the way, and the computation of the corresponding Delaunay graph

(Delaunay tessellation) follows as a simple consequence. The time complexity of the algorithm is bounded (not necessarily tightly) above by an expression of the form $O(n^2)$. This is better than the time complexity of the naive method and equals the one of the incremental method [29], [38], [39, pp. 242-251]. As mentioned, this complexity is not known to be tight and the actual behavior is however more or less linear, better than the expected $O(n \log(n))$ behavior of the incremental method [30].

1.3. The structure of the paper. In Section 2 the notation, terminology and several tools are introduced. In Section 3 a schematic description of the algorithm is given. A detailed description of the algorithm in dimension 2 is given in Section 4. A method for finding endpoints in an exact way is described in Section 5. The method of storing the data as well as a discussion on some combinatorial issues are described in Section 6. In Section 7 it is explained how to compute the Delaunay graph. A discussion on several theoretical and practical issues is given in Section 8. Proofs of some claims are given in Section 9.

2. PRELIMINARIES

In this section we present the notation and basic definitions used later, as well as some helpful tools. Our world X is a convex and compact polygon (of dimension m) in the Euclidean space $(\mathbb{R}^m, |\cdot|)$. The induced metric is $d(x, y) = |x - y|$. We denote by $[p, x]$ and $[p, x)$ the closed and half open line segments connecting p and x , i.e., the sets $\{p + t(x - p) : t \in [0, 1]\}$ and $\{p + t(x - p) : t \in [0, 1)\}$ respectively. The inner product between the vectors $x = (x_1, \dots, x_m)$ and $y = (y_1, \dots, y_m)$ is $\langle x, y \rangle = \sum_{i=1}^m x_i y_i$. A nonnegative linear cone emanating from a point p and generated by the vectors t_1, \dots, t_m is the set $\{p + \sum_{i=1}^m \lambda_i t_i : \lambda_i \geq 0 \forall i\}$. A line in \mathbb{R}^2 , a plane in \mathbb{R}^3 , or, more generally, a hyperplane in \mathbb{R}^m , are denoted by L, M , etc. A face of a multidimension polygon located on a corresponding hyperplane L is denoted by \tilde{L} . In dimension 2 a face of the cell is just an edge and we alternatively use the names “edge”, “side”, “face”, or “facet” for describing this notion in this dimension. The sites are denoted by P_k , $k \in K = \{1, \dots, n\}$. They are assumed to be points and $P_k \neq P_j$ whenever $k \neq j$.

We note that the definitions and results described below hold under more general assumptions. However, in order to avoid apparent complications, we confine ourselves to the setting of point sites located in a convex polygonal region contained in a finite dimensional Euclidean space.

Definition 2.1. *The Voronoi diagram of the tuple of point sites $(P_k)_{k=1}^n$ contained in the region X is the tuple $(R_k)_{k=1}^n$ of subsets $R_k \subseteq X$ where, for each $k \in K = \{1, \dots, n\}$,*

$$R_k = \{x \in X : d(x, P_k) \leq d(x, P_j) \quad \forall j \in K, j \neq k\}.$$

In other words, the Voronoi cell R_k associated with the site P_k is the set of all $x \in X$ whose distance to P_k is not greater than their distance to the other sites P_j .

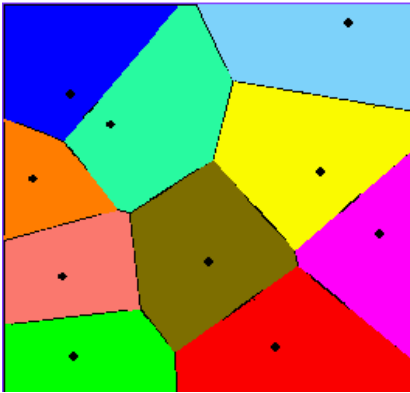


FIGURE 1. A Voronoi diagram of 10 point sites in a square in the Euclidean plane.

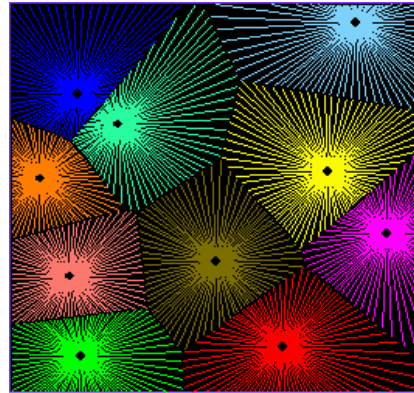


FIGURE 2. Each of the cells of Figure 1 is approximated using 80 rays.

The definition of the Voronoi diagram is analytic. However, it can be easily seen that each cell R_k is the intersection of X with the halfspaces $\{x \in \mathbb{R}^m : d(x, P_k) \leq d(x, P_j), j \neq k\}$. Thus each cell is a (multidimensional) closed and convex set which can be represented using its combinatorial structure, namely its ℓ -dimensional faces $\ell = 0, 1, 2, \dots$ (the vertices, sides, etc.). Because of this property the traditional approach to Voronoi diagrams is combinatorial.

In a recent work [42], a different representation of the cells was introduced, suggesting to consider each of the cells as a union of rays (lines segments):

Theorem 2.2. *The Voronoi cell R_k of a site $p = P_k$ is a union of rays emanating from p in various directions. More precisely, denote $A = \bigcup_{j \neq k} P_j$. Given a unit vector θ , let*

$$T(\theta, p) = \sup\{t \in [0, \infty) : p + t\theta \in X \text{ and } d(p + t\theta, p) \leq d(p + t\theta, A)\}. \quad (2.1)$$

The point $p + T(\theta, p)\theta$ is the endpoint corresponding to the ray emanating from p in the direction of θ . Then

$$R_k = \bigcup_{|\theta|=1} [p, p + T(\theta, p)\theta].$$

This representation actually holds (after simple modifications) in a more general setting (any norm, any dimension, sites of a general form, etc.) and it shows that by “shooting” enough rays one can obtain a fairly good approximation of the cells (see Figures 1-2). However, as explained in Section 1, it is not immediate to obtain the combinatorial structure from this representation. Nevertheless, it will be shown in later how the idea of shooting rays can indeed be used for obtaining this structure.

3. A SCHEMATIC DESCRIPTION OF THE ALGORITHM

In this section we present a schematic description of the algorithm for computing the Voronoi cells. The method is based on the fact that the cell of some point site

$p = P_k$ is a convex polygon whose boundary consists of vertices and edges. A detailed description for the case of dimension $m = 2$, including a pseudo code and illustrations, are given in Section 4. Some of the involved concepts are in the spirit of optics and convex analysis. However, we use them sometimes in a way different from the familiar one. For instance, although in Method 3.1 we use a simplex and the point p is an interior point, there is no connection between our method and the so-called simplex method and interior point methods used for solving convex optimization and (non)linear programming problems [12, 21].

A formulation of the method is described below. Recall again that for a unit vector θ , the point $p + T(\theta, p)\theta$ is the endpoint corresponding to the ray emanating from p in the direction of θ .

Method 3.1. *A high level description:*

- **Input:** A site p ;
 - **Output:** The (combinatorial) Voronoi cell of p ;
- (1) Think of p as a light source;
 - (2) emanate a (linear) conic beam of light from p using a simplex;
 - (3) detect iteratively (by possibly dividing the cone to suncones) all possible vertices (and additional related combinatorial information) inside this beam using corresponding endpoints and an associated system of equations;
 - (4) continue the process with other beams until the whole space around p is covered;

The actual generation and handling of the (sub)cones is done using a simplex located around p . The boundary of this simplex is initially composed of $(m-1)$ -dimensional faces, and later these faces are composed of subfaces, when we narrow the search to subcones (sub-beams). Each such a subface induces a cone: the cone generated by the rays which pass via the corners of the subface; See Figure 5. Each such a corner induces a unit vector (denoted by θ) which points in its direction (from p). Once the corresponding unit vector is known, then so is the ray in its direction.

The system of equations mentioned above (Step (3)) is

$$B\lambda = H, \tag{3.1}$$

where the vector of unknowns is $\lambda = (\lambda_1, \dots, \lambda_m)$, B is the m by m matrix with entries $B_{ij} = \langle N_i, T_j \rangle$ and H is an m -dimensional vector with entries $H_i = \langle N_i, T_i \rangle$, $i, j = 1, 2, \dots, m$. The notation $T_i = T(\theta_i, p)\theta_i$ means the vector in direction θ whose length is the distance from p to the endpoint $p + T_i$. The m -dimensional vector N_i is a normal to the hyperplane $L_i = \{x : \langle N_i, x \rangle = c_i = \langle N_i, p + T_i \rangle\}$ on which the endpoint $p + T_i$ is located.

Equation (3.1) has a simple geometric meaning: the point $u = p + \sum_{i=1}^m \lambda_i T_i$ is in the intersection of the hyperplanes L_1, \dots, L_m if and only if λ solves (3.1). If we want to restrict ourselves to the cone generated by the corresponding rays, then we consider only the nonnegative solutions of (3.1), i.e., $\lambda_i \geq 0$ for all $i = 1, \dots, m$. If equation (3.1) has a unique nonnegative solution λ , then this means that u is a point in the cone which is a candidate to be a vertex of the cell, since it may be (but is not

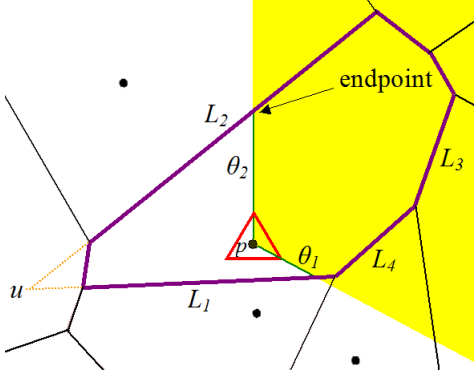


FIGURE 3. Illustration of the algorithm. The cone generated by the subface $\{\theta_1, \theta_2\}$ is shown. The intersection between L_1 and L_2 is a point outside the cone and hence the cone is divided. The next two subfaces are $\{\theta_1, \theta_3\}$, $\{\theta_2, \theta_3\}$.

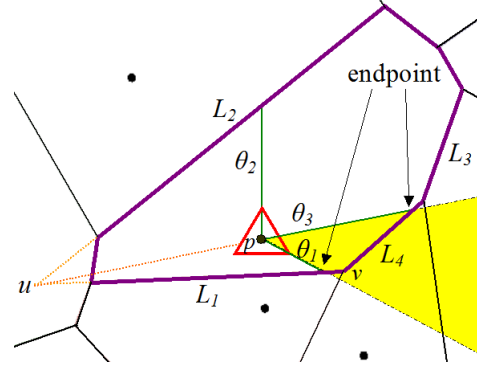


FIGURE 4. Now the cone generated by the subface $\{\theta_1, \theta_3\}$ is shown. Since $v = L_1 \cap L_4$ is a point in the cone and the cell it is a vertex and no further dividing of this cone is needed.

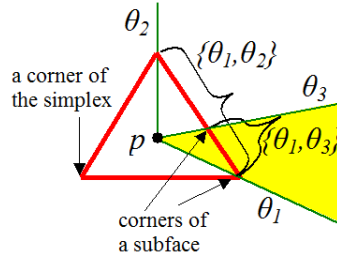


FIGURE 5. The simplex, some of its subfaces, and the conic beam emanating from p and corresponding to the subface $\{\theta_1, \theta_3\}$.

necessary) in the intersection of the corresponding m different facets located on the hyperplanes L_i . If, in addition, u is known to be in the cell, then it is indeed a vertex.

4. A DETAILED DESCRIPTION FOR THE CASE OF DIMENSION $m = 2$

Method 3.1 in its full description can be best understood for dimension $m = 2$, and in this section we consider only this case. See page 8 for a pseudo-code and Figures 3-4 for an illustration. Higher dimensional cases are more complicated because of Step (3), and they will be discussed elsewhere. Recall again that in dimension 2 a face of the (boundary of the) cell is just an edge.

In what follows we will explain the method and the pseudo-code in more details. First, we create the 3 unit vectors θ_i corresponding to a simplex (a triangle) around the point p . After choosing a simplex subspace, shooting the two rays in the direction

Algorithm 1: The 2D algorithm: a detailed pseudocode

input : A site p whose cell is to be computed
output: The vertices and edges of the cell, other information

- 1 Create the simplex unit vectors;
- 2 Create the simplex faces and enter them into *FaceQueue*;
- 3 **while** *FaceQueue* is nonempty **do**
- 4 Consider the highest (first) subface in *FaceQueue*;
- 5 Denote it by $\{\theta_1, \theta_2\}$;
- 6 Compute the endpoints $p + T_i$, $i = 1, 2$;
- 7 Find their neighbor sites a_i , $i = 1, 2$;
- 8 Compute the bisecting line L_i between p and a_i , $i = 1, 2$;
- 9 If no such a site a_i exists, then $p + T_i$, is on the boundary of the world. Call the corresponding boundary line L_i ;
- 10 Consider the system of equations (3.1)
- 11 **if** $\det(B) = 0$ **then** // no solution or ∞ many
- 12 **if** $L_1 = L_2$ **then** // no vertices here
- 13 continue;
- 14 **else** // parallel lines
- 15 $\theta_3 = \phi$ where ϕ is the direction vector of the lines;
- 16 If the ray in the direction of ϕ is not in the cone, then $\theta_3 = -\phi$;
- 17 Insert the subfaces $\{\theta_1, \theta_3\}$, $\{\theta_2, \theta_3\}$ into *FaceQueue*;
- 18 **else** // $\det(B) \neq 0$, unique solution λ
- 19 $u = p + \lambda_1 T_1 + \lambda_2 T_2$;
- 20 **if** λ is nonnegative **then** // we're in the cone
- 21 **if** u is inside the cell **then**
- 22 Store u , L_1, L_2 (and/or neighbor sites;) // u is a vertex
- 23 **else** // u is outside the cell
- 24 $\theta_3 = (u - p)/|u - p|$;
- 25 Insert $\{\theta_1, \theta_3\}$, $\{\theta_2, \theta_3\}$ into *FaceQueue*;
- 26 **else** // u isn't in the cone
- 27 $\theta_3 = (p - u)/|p - u|$;
- 28 Insert $\{\theta_1, \theta_3\}$, $\{\theta_2, \theta_3\}$ into *FaceQueue*;
- 29 Remove $\{\theta_1, \theta_2\}$ from *FaceQueue*;

of θ_i , $i = 1, 2$, finding the endpoints $p + T_i$ (using, e.g., Method 5.1 in Section 5; here $T_i = T(\theta_i, p)\theta_i$) and finding the corresponding bisecting lines L_i , we want to use this information for finding all of the possible vertices in the cone generated by the rays. By using equation (3.1) we find the type of intersection between the lines L_1, L_2 . This intersection is either the empty set, a point, or a line.

If (3.1) has no solution of any kind (including solutions which are not non-negative), then the lines L_1 and L_2 are parallel. This is a rare event but it must be taken into account. In this case L_1 and L_2 have the same direction vector ϕ , i.e., $L_i = \{q_i + \phi t : t \in \mathbb{R}\}$ for some $q_i \in \mathbb{R}^2$, $i = 1, 2$ and some unit vector ϕ . We check if the ray

emanating from p in the direction of ϕ is in the cone (happens if and only if the solution (α_1, α_2) to the linear equation $\phi = \alpha_1\theta_1 + \alpha_2\theta_2$ is nonnegative). If yes, then we shot a ray in the direction of $\theta_3 := \phi$. Otherwise, we shot the ray in the direction of $\theta_3 = -\phi$. This ray will be contained in the cone and will hit a facet of the cell not located on the lines L_1 and L_2 (the facet may be located on the boundary of the bounded world X). We divide the current simplex subface using θ_3 and continue the process.

If $L_1 = L_2$, then both endpoints are located on the same line. This corresponds to the case where (3.1) has infinitely many solutions. In this case there is no vertex in the corresponding cone (perhaps one of the endpoints $p + T_i$ is a vertex, but this vertex will be found later using the neighbor subface). Hence we can finish with the current subface and go to the other ones. Such a case is implicit in Figure 4 when the rays are shot in the directions of the first and third corners of the simplex and hit L_1 .

If (3.1) has a unique solution $\lambda = (\lambda_1, \lambda_2)$, then either it is not nonnegative, i.e., the point $u = p + \sum_{i=1}^2 \lambda_i T_i$ is not in the cone, or λ is nonnegative, i.e., u is in the cone. In the first case the ray emanating from p in the direction of $\theta_3 := (p - u)/|p - u|$ will hit an edge of the cell contained in the cone but not located on L_1 or L_2 . Such a case is described in Figure 4 when considering the rays of θ_1 and θ_2 . We divide the current simplex subface using θ_3 and continue the process. In the second case u is in the cone, but we should check if u is in the cell (can be checked, for instance, by distance comparisons). If u is in the cell (corresponding to the case of θ_1 and θ_3 in Figure 4), then it is a vertex and we store it (together with other data: see Section 6). We have finished with the subface and can go to the other ones. Otherwise u is not a vertex, and we actually found a new edge of the cell corresponding to the ray in the direction of $\theta_3 := (u - p)/|u - p|$ (implicit in Figure 4 when considering the rays shot in the directions of the second and third corners of the simplex; u is the intersection of L_1 and L_2). We divide the subface using θ_3 and continue the process.

From the above description it seems that the method should be implemented in a recursive way. However, by using a simple data structure, one can avoid the need to use a recursive implementation and instead can use loops. The reason that this is possible is because each subface is handled independently of the other subfaces, including its “parent” or “children”: no information is exchanged between the subfaces. As a result, one can maintain a list of subfaces, called *FaceQueue* (each subface is represented by a set of two unit vectors, which correspond to its corners), and run the process until this list is empty. The initial list contains the faces of the simplex $\{\phi_1, \phi_2\}$, $\{\phi_2, \phi_3\}$, $\{\phi_1, \phi_3\}$, where we can take $\phi_1 = (\sqrt{3}/2, -1/2)$, $\phi_2 = (0, 1)$, $\phi_3 = (-\sqrt{3}/2, -1/2)$. In this connection, it should be emphasized that the simplex is used for handling the progress of the algorithm (using the list of subfaces), but the corresponding unit vectors in the direction of the subfaces’ corners are not necessarily on the same line as the one on which the simplex subface is located. Despite this, it is convenient to represent a simplex subface by its associated unit vectors.

5. FINDING THE ENDPOINTS EXACTLY

In order to apply Method 3.1, we should be able to find the endpoint $p + T(\theta_i, p)\theta_i$ emanating from the site p in the direction of θ_i [see (2.1) and line (6) in Algorithm 1]. One possible method is to use the method described in [42], but the problem is that the endpoint found by this method is given up to some error parameter, and unless this parameter is very small (which, in this specific case, implies slower computations), this may cause an accumulating error later when finding the vertices, due to numerical errors in the expressions in equation (3.1).

In what follows we will describe a new method for finding the endpoint in a given direction θ exactly. Of course, when using floating point arithmetic errors appear, but they are much smaller than the ones described above. The method can be implemented in any dimension. See Figure 6 for an illustration in dimension 2.

Method 5.1.

- **Input:** A site p and a unit vector θ ;
 - **Output:** the endpoint $p + T(\theta, p)\theta$.
- (1) Shoot a ray from p in the direction of θ and stop it at a point y which is either in the region X but outside the cell of p , or it is the intersection of the ray with the boundary of the region. If y is chosen to be outside the cell, then go to Step (4). Otherwise, let L be the boundary hyperplane on which y is located;
 - (2) check whether y is in the cell, e.g., by comparing $d(y, p)$ to $d(y, a)$ for any other site a , possibly with enhancements which allow to reduce the number of distance comparisons;
 - (3) if y is in the cell, then y is the endpoint and L is the bisecting hyperplane. The calculation along the ray is complete;
 - (4) otherwise, $d(y, a) < d(y, p)$ for some site a . Let $CloseNeighbor := a$;
 - (5) find the point of intersection (call it u) between the given ray and the bisecting hyperplane L between p and $CloseNeighbor$. This intersection is always nonempty. The hyperplane is easily found because it is vertical to the vector $p - CloseNeighbor$ and passes via the point $(p + CloseNeighbor)/2$;
 - (6) let $y := u$; go to Step (2).

Remark 5.2. The correction of this method is quite simple. First, the method terminates after finitely many steps because there are finitely many sites. The point u in Step (5) is well defined because y is in the halfspace of $CloseNeighbor$ and hence the considered ray intersects the boundary L of this halfspace. The point y is outputted in Step (3) and by the description of this step y is in the cell. Since y is on a bisecting hyperplane between p and another site, and since y is in the cell, this means that y must be an endpoint (see (2.1); here $A = \cup_{j \neq k} P_j$ and $T(\theta, p) = |y - p|$).

Remark 5.3. When finding the endpoint y , one can also find all of its neighbor sites since in the last time Step (2) is performed, one can easily find all the sites a satisfying $d(a, y) = d(p, y)$ simply by storing any site a satisfying this equality. Call the corresponding list *EquiDistList*. Any $a \in EquiDistList$ induces a corresponding bisecting hyperplane L between p and a . Suppose from now on that the dimension

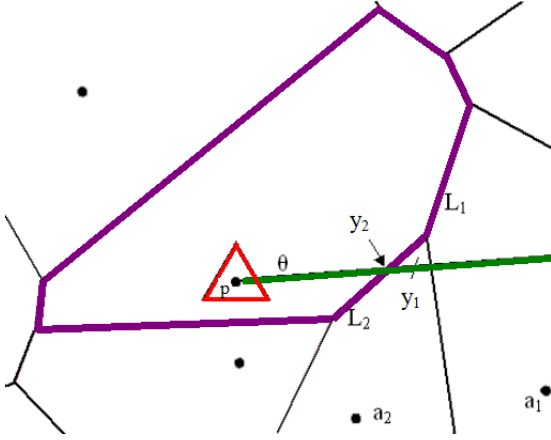


FIGURE 6. Illustration of Method 5.1 for some ray. The dimension of the world is 2. The ray comes from far away. At the first displayed iteration *CloseNeighbor* is a_1 . The intersection between the corresponding line L_1 and the ray is $y = y_1$. At the next stage *CloseNeighbor* is a_2 and $y = y_2$. The process terminates since y_2 is in the cell of p , i.e., it is the endpoint.

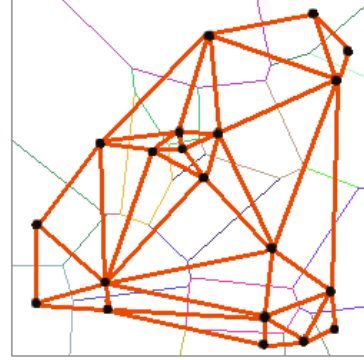


FIGURE 7. The Delaunay graph of 20 sites, restricted to a square in the plane (thick line). The Voronoi cells of the same sites are shown too (thin line).

is 2. In the rare event where y coincides with a vertex of the cell, there may be $a \in \text{EquiDistList}$ whose bisector L may not contain a facet of the cell but rather it intersects the cell only at the vertex y (this can happen only when *EquiDistList* has at least 3 different elements, and hence does not happen when the sites are in general position). This is a problem, since we want to make sure that when we make operations with the endpoint, we use a line L on which it is located and on which an edge of the cell is located. In order to overcome the problem, one simply needs to know which of the several sites $a \in \text{EquiDistList}$ induces a bisector L containing an edge of the cell, and then to consider this site and its associated line for later operations. This can be easily done by sorting in increasing order all the angles $\angle pya$, $a \in \text{EquiDistList}$, or, equivalently, the distances $d(a, p)$, $a \in \text{EquiDistList}$ (the equivalence is because p and all the sites $a \in \text{EquiDistList}$ are on the circle whose center is y). The sites corresponding to the two smallest values are the ones which induce a desired bisector.

6. STORING THE DATA AND A NEW COMBINATORIAL REPRESENTATION

In this section we explain how the information retrieved by Method 3.1 is stored. Along the way a new combinatorial representation of the Voronoi cells is presented and the difference between this representation and familiar ones is explained.

Given a point site $p = P_k$, when a vertex u from the cell of p is found, one stores the following features: its coordinates, the hyperplanes from which it was obtained (i.e.,

u belongs exactly to the corresponding faces located on these hyperplanes), and the index k . For storing a hyperplane L it is convenient to store the index of its associated neighbor site, namely the index (simply a number or a label) of the site which induces it (denoted by *CloseNeighbor* in Method 5.1). If it is a boundary hyperplane, then it has a unique index number which is stored and from this index one can retrieve the parameters (the normal and the constant) defining the hyperplane. Alternatively, these parameters can be stored directly. For some purposes it may be useful to store also some endpoints associated with each hyperplane. A convenient data structure for storing the whole diagram is a one dimensional array, indexed k , in which the vertices (represented, as explained above, by coordinates and associated neighbor sites) and any additional information, such as endpoints, are stored.

In dimension $m = 2$ a vertex is always obtained from 2 lines. In dimension $m \geq 3$ a vertex is usually obtained from exactly m hyperplanes, but in principle it can be obtained from S hyperplanes, $S > m$. We call the set $\{L_{i_1}, \dots, L_{i_S}\}$ of all the hyperplanes from which u was obtained the combinatorial representation of u .

As the examples below show (see also Section 7), once the above information is stored, we can obtain other combinatorial information related to the cell, say the neighbors of a given vertex, the edges of some cell, the Delaunay graph, and so on, and hence we do not need to store these types of information separately. This is in contrast with familiar methods for representing the combinatorial information in which one has to find and store all the ℓ -dimensional faces, $\ell = 0, 1, \dots, m - 1$ of the cell. Since the combinatorial complexity of the cell (the number of multi-dimensional faces) may grow exponentially with the dimension [6, 34], our method may save a lot of space.

There is another difference between our method and other ones. In other method one starts with the vertices as the initial (atomic) ingredients, and from them one constructs higher dimensional faces. For instance, an edge is represented by the two vertices which are its end points. However, in our method we start with the highest dimensional faces (located on hyperplanes) and from them we construct the vertices and the other multi-dimensional faces. As a matter of fact, our method of storage can be used to represent any multi-dimensional convex polygon, and even a class of nonconvex or abstract ones.

Example 6.1. The possible ℓ -dimensional faces, $\ell = 0, 1, \dots, m - 1$, can be found by observing that any such a face is the intersection of hyperplanes. Indeed, the $(m - 1)$ -dimensional faces are all the hyperplanes L_i which appear in the representation of the vertices. For finding the $(m - 2)$ -dimensional faces we fix an hyperplane L_i and look at all the vertices u having L_i in their combinatorial representation. In the representation of any such a vertex u appear other hyperplanes L_j , and $L_i \cap L_j$ is an $(m - 2)$ -dimensional face. By going over all the possible hyperplanes of u , all the possible u , and all the possible L_i , we can find and represent all the possible $(m - 2)$ -dimensional faces. A similar procedure can be used for the other ℓ -dimensional faces.

Example 6.2. Ordering the vertices in clockwise or counterclockwise order is easily done by the method of search, e.g., by labeling the rays with corresponding values.

Example 6.3. Given a vertex u with a given combinatorial representation, its neighbor vertices can be found by observing that if u and v are neighbors, then they are located on the same 1-dimensional face. This face is the intersection of $m - 1$ hyperplanes. Hence for finding the neighbor vertices of u we simply need to go over the list of vertices v and choose the ones whose combinatorial representation contains $m - 1$ hyperplanes which also appear in the representation of u .

7. COMPUTING THE DELAUNAY GRAPH (DELAUNAY TESSELLATION)

Here it is shown how to compute the Delaunay graph of the given sites in an almost automatic way from the data structure used for storing the Voronoi diagrams.

The Delaunay graph is an important geometric structure which by itself has many applications [6, 22, 39]. By definition, the Delaunay graph consists of vertices and edges. The vertices of the Delaunay graph are the (point) sites. There is an edge between two sites if their Voronoi cells are neighbors (via a face). See Figure 7 for an illustration in dimension 2. Note that here everything is restricted to the given world X . Rarely it may happen that two sites whose cells are neighbors in the whole space are not neighbors in X . This can happen only with cells which intersect the boundary of X . For overcoming this problem (if one considers this as a problem) one can simply take X to be large enough or can perform a separate check for the above boundary cells.

As a result, for computing the Delaunay graph one chooses a given site, goes over the data structure used for the Voronoi cells (see Section 6), and finds all the different neighbor sites of a given site. The procedure is repeated for each site. For drawing the Delaunay graph one simply connects two sites by an edge when it is found that one site is a neighbor site to another one. The above procedure can be easily implemented in parallel.

8. SEVERAL THEORETICAL AND PRACTICAL ASPECTS

The following theorem describes several aspects related to the algorithm.

Theorem 8.1. *Suppose that the world $X \subset \mathbb{R}^2$ is a compact and convex subset whose boundary is polygonal. Assume also that the distinct point sites P_1, \dots, P_n are contained in its interior. Then Algorithm 1 is correct. More specifically, given any site $p = P_k$, the computation of the Voronoi cell of p by Algorithm 1 terminates after a finite number of steps and the corresponding entries in the output of the algorithm include all the vertices and edges of the cell. Its time complexity is bounded above (not necessarily tightly) by an expression of the form $O(n^2)$.*

The proof of Theorem 8.1 is quite technical (see Section 9), but the main idea regarding the bound on the time complexity is simple: each time a ray is shot, a new edge of the cell is detected or a vertex is found. This shows (after careful counting) that the number of rays used for each cell is bounded by a universal constant times the number of edges in each cell. Hence the total number of edges is of the order of the size of the diagram. It is well known that this size is $O(n)$ (see [6, p. 347], [40, pp. 173-5]). Since the number of operations done in each ray (distance comparisons)

is $O(n)$ (or $O(1)$ with high probability if the points are generated according to the uniform distribution), the bound $O(n^2)$ follows (or $O(n)$ with high probability for the uniform distribution case).

It is not clear whether the bound on the time complexity presented in Theorem 8.1 is tight, after possibly taking into account several enhancements but we believe that such enhancements can give better bounds (at least in the one processor case): e.g., perhaps by computing the cells in a certain order (e.g., using a sweep line), by improving the endpoint computation (Method 5.1), etc. We do have several experimental results and partial theoretical explanations in the special but important case where the sites are generated by the uniform distribution. If a preprocessing stage (whose complexity is $O(n)$) is performed in which the sites are inserted into a corresponding data structure (buckets) mentioned in [10], then the number of distance comparisons needed for computing the endpoints of the rays can be significantly reduced. In this case the uniform distribution of the sites ensures, with high probability, that only very near sites will be considered for the distance comparisons. This results in time complexity of $O(1)$ for each cell and $O(n)$ for the whole diagram. The difference between the potential worst case scenarios and the practical one is similar in some sense to the Quicksort algorithm for sorting [19, 32] whose average time complexity is $O(n \log(n))$ but its worst case time complexity is $O(n^2)$. The simplex algorithm for linear programming [21] provides another similar example (efficient in practice, with polynomial time average case complexity, but exponential time complexity in the worst case [35]).

The actual behavior of our implementation is somewhat strange: for a reason which is not currently well understood, the running time sometimes grows in a way which is a little bit greater than linear with respect to the input (number of sites) when one processing unit is involved. However, when several processing units are involved, then the running time $t(n)$ is better than linear with respect to the input (i.e., $t(\alpha n) < \alpha t(n)$ for any tested $\alpha \in \mathbb{N}$; this does not contradict the obvious lower bound of $O(n)$). Perhaps (in both cases) this may be related to something in the memory management or some influence of the operating system.

Comparing with well-known implementations, our preliminary implementation performs quite good. For example, it runs faster than Qhull 2011.1 [9] when a Voronoi diagram of 10^6 sites is computed (a few seconds on a standard computer). However, it does not run faster than Triangle [48, 49]. Both Qhull and Triangle are veteran serial implementations (more than 15 years) which have adopted many enhancements over the years. Despite this, their behavior is worst than linear and their output is not always correct. In contrast, in our implementation only two people have been involved (only one of them has done the programming work) and many enhancements are waiting to be implemented. For instance, each vertex usually belongs to 3 cells and hence it is computed 3 times, while in Qhull/Triangle it is computed only once. In addition, at least in the multiprocessor case, it is better than linear, and so far no incorrect output has been observed when double precision arithmetic was used. A detailed description of experimental results and more details about implementation issues in various environments will be discussed elsewhere.

9. PROOF OF THEOREM 8.1

In this section we prove Theorem 8.1 which shows the correction of the algorithm and presents an upper bound on its time complexity. The proof is based on several lemmas. Here are additional details regarding notation and terminology. In the sequel $p = P_k$ is a given site. We assume that the sites are different, and hence $\min\{d(P_k, P_j) : k \neq j\} > 0$. Since the distance from p to the boundary of the cell is positive, there is a small circle with center p which is contained in the interior of the cell. On this circle we construct the simplex, which, as a consequence, has a positive distance from the boundary of the cell and from p . Any other simplex which may be used in the algorithm for producing the rays and detecting the vertices is simply a scalar multiplication of this small simplex and it produces exactly the same rays (hence enables to detect the same vertices). Recall again that we use the notion “face” or “facet” for denoting an edge of the cell (a 1-dimensional face).

A vertex v of the cell is said to correspond to a subface F of the simplex if v is located inside the cone corresponding to F . For instance, in Figure 4 the vertex v corresponds to the subface $F = \{\theta_1, \theta_3\}$ and no other vertices correspond to F . All the rays we consider emanate from a given site $p = P_k$ and are sometimes identified with their direction vectors. Given a cone generated by two rays, the rays on the boundary of the cone are called boundary rays and any other ray in the cone is called an intermediate ray. For instance, in Figure 4 if we look at the cone generated by θ_1 and θ_2 , then these rays are boundary rays and θ_3 is an intermediate ray. Continuing with Figure 4, after additional steps additional rays will be generated between θ_1 and θ_3 , and hence they will be intermediate rays in the cone generated by θ_1 and θ_2 (and also of the one generated by θ_1 and θ_3). Between θ_1 and θ_3 no additional rays will be generated.

Part of the proof is to analyze the time complexity of the algorithm and in particular proving that the algorithm terminates after finitely many steps. For doing this it is convenient to consider its tree, namely the graph whose nodes are the main stages in the algorithm, and a node has subnodes (children nodes) whenever a conditional operation (separating into cases, if-else, etc.) is made in the node whose result is a non-terminating-condition statement (i.e., further work is required by the algorithm to achieve a terminating condition).

In Algorithm 1 there are two types of nodes: in the first type (lines 12 and 21) the algorithm finishes its consideration with the current subface (perhaps after storing some information) without further dividing it. In the second type (lines 14, 23, and 26) the subface is divided into two subfaces and the algorithm continues with both of them. In other words, in the first type there are no children nodes and in the second one there are. For the sake of simpler analysis the computation of the two endpoints using (3.1) (or passing them in a computed form from previous stages) done before each conditional operation is considered as performed in each node and not in a separated node. If, for the sake of fully counting the number of nodes, such a computation is considered in a separate node, then the total number of nodes will

be greater by at most a factor of 2 than in our analysis since before each of the two types of nodes mentioned above there is only one “endpoint node”.

Lemma 9.1. *Consider a simplex subface $F = \{\theta_1, \theta_2\}$ and the corresponding cone generated from it. Let L_1 and L_2 be the lines on which the endpoints $p + T(\theta_i, p)\theta_i$, $i = 1, 2$ are located. Suppose that an intermediate ray in the direction of θ_3 is generated when F is considered. Then the ray of θ_3 hits a facet different from \widetilde{L}_i , $i = 1, 2$.*

Proof. First note any ray generated by the algorithm does hit a facet \widetilde{L}_3 . Indeed, the ray starts at a point inside the cell and goes outside the cell (because the region X is bounded), so the intermediate value theorem implies that this ray intersects the boundary of X , namely, a certain facet \widetilde{L}_3 .

Denote by g the hitting point. This point is in fact the endpoint. Indeed, by the definition of the endpoint (see (2.1)) the ray of θ_3 has an endpoint $p + T(\theta_3, p)\theta_3$. Any point on the ray after g is strictly outside the cell (either because it is outside the world or because it is in a halfspace of another site). Thus $p + T(\theta_3, p)\theta_3 \in [p, g]$. However, any point beyond $[p, p + T(\theta_3, p)\theta_3]$ is outside the cell by the definition of the endpoint. Since we know that g is in the cell (it belongs to the facet \widetilde{L}_3) we obtain that $g \in [p, p + T(\theta_3, p)\theta_3]$. Thus $g = p + T(\theta_3, p)\theta_3$.

Now we observe that an intermediate ray is created only when either L_1 and L_2 are parallel (line 14), or when they intersect in the cone but outside the cell (line 23), or when they intersect outside the cone (line 26).

In the first case the line on which the ray of θ_3 is located is parallel to L_1 and L_2 , and hence cannot hit them. In particular $\widetilde{L}_3 \neq \widetilde{L}_i$, $i = 1, 2$.

Now consider the second case and suppose for a contradiction that, say, $\widetilde{L}_1 = \widetilde{L}_3$. Let $u = L_1 \cap L_2$. By assumption u is outside the cell. In particular u is hit by the ray of θ_3 (see line 23) and $u \neq p + T(\theta_3, p)\theta_3$. But both u and $p + T(\theta_3, p)\theta_3$ are assumed to belong to \widetilde{L}_1 and hence to L_1 . Thus L_1 coincides with L_3 and in particular passes via p . This is impossible since p is in the interior of the cell and hence its distance to any of the boundary lines is positive. The third case is handled similarly to the second one. \square

Lemma 9.2. *Consider two different rays generated by the algorithm, say in the direction of ψ_1 and ψ_2 . These rays may belong to different simplex subfaces but both of them are assumed to be between (and possibly coincide with) two initial rays, i.e., rays induced by two corners of the simplex. Consider an intermediate ray between the rays of ψ_i , $i = 1, 2$ whose direction vector is θ_3 . Then the endpoint of the ray of θ_3 and the endpoints of the rays of ψ_1 and ψ_2 must be located on different facets.*

Proof. Suppose to the contrary that this is not true, say the endpoint of the ray of ψ_1 and the endpoint of the ray of θ_3 are located on the same facet \widetilde{L} . Because the ray of θ_3 is between two initial rays it is generated from some subface $\{\theta_1, \theta_2\}$. Here θ_1 is between ψ_1 and θ_3 and possibly equals ψ_1 , and θ_2 is between θ_3 and ψ_2 and possibly equals ψ_2 . This generation corresponds to the routines of lines 14, 26, or 23 of the algorithm and in particular $\theta_1 \neq \theta_3$ and $\theta_2 \neq \theta_3$. Denote by L_i , $i = 1, 2$ the lines on which the endpoints corresponding to θ_i , $i = 1, 2$ are located.

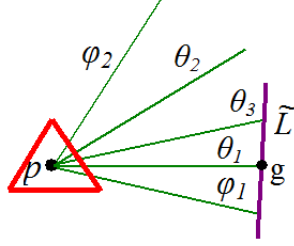


FIGURE 8. An illustration of Lemma 9.2.

The ray of θ_1 intersects \tilde{L} at some point g which is its endpoint as explained in the proof of Lemma 9.1. See also Figure 8). We conclude that the endpoints of the rays of θ_3 and of θ_1 are located on the same facet. But this is impossible, since by assumption θ_3 was created from the simplex subface $\{\theta_1, \theta_2\}$ and as already mentioned, this can happen only when either L_1 and L_2 are parallel (line 14), or when they intersect outside the cone (line 26) or when they intersect in the cone but outside the cell (line 23), and in all of these cases the ray of θ_3 hits a facet different from the ones associated with θ_1 and θ_2 (see Lemma 9.1). We arrived to a contradiction which proves the assertion. \square

Lemma 9.3. *The algorithm terminates after a finite number of steps. In particular, the number of intermediate rays is finite.*

Proof. This follows from Lemma 9.2. Indeed, suppose for a contradiction that the algorithm does not terminate after finitely many steps. This means that the list of simplex subfaces *FaceQueue* is never empty. But in each stage of the algorithm (each node), either a subface is divided (lines 14, 23, and 26) and then replaced by its children subfaces and deleted from *FaceQueue*, or it has no children subfaces (lines 12 and 21) and hence it is deleted from *FaceQueue* shortly after its creation. After a subface is deleted from *FaceQueue* it is never created again. As a result, the fact that the algorithm does not terminate after finitely many steps implies that we can find an infinite sequence of nested distinct subfaces. A subface is created only when a new edge of the cell is detected (see again lines 14, 23, and 26). Such an edge is created using an intermediate ray and hence could not be detected before because of Lemma 9.2. Thus infinitely many distinct edges are detected, contradicting the fact that each cell has only finitely many edges. Hence the algorithm terminates after finitely many steps and in particular finitely many intermediate rays are created. \square

Lemma 9.4. *Given a cone generated by two linearly independent unit vectors θ_1 and θ_2 , let E be the number of facets of the cell hit by intermediate rays, i.e., rays produced by the algorithm inside the cone excluding the boundary rays of the cone. Then the tree of the restriction of the algorithm to the given cone contains exactly $2E + 1$ nodes.*

Proof. The proof is by induction on E . By Lemma 9.3 we know that E is finite. Let \tilde{L}_i be the facet corresponding to θ_i , $i = 1, 2$. If $E = 0$, then there are two cases. In

the first case the rays generated by θ_i , $i = 1, 2$ hit the same facet and in this case (line 12) the current simplex subface is deleted from *FaceQueue*. Thus no subnode of the current node is created, i.e., the restriction of the algorithm tree to the cone contains $1 = 2E + 1$ nodes. In the second case the rays must hit different facets which intersect at a vertex, since otherwise either no intersection occurs or the intersection is a point outside the cell or outside the cone, and the corresponding ray in the direction of θ_3 (lines 14, 23, 26) hits a facet contained in the cone, contradicting $E = 0$. Therefore also in this case the current subface is deleted from *FaceQueue* and no subnode of the current node is created and the restriction of the algorithm tree to the cone contains $2E + 1$ nodes.

Now assume that the claim holds for any nonnegative integer not exceeding $E - 1 \geq 0$. It will be proved that it holds for E too. Indeed, if the current simplex subface $\{\theta_1, \theta_2\}$ is not divided into two subfaces $\{\theta_1, \theta_3\}$ and $\{\theta_2, \theta_3\}$, then the restriction of the algorithm tree to the cone generated by $\{\theta_1, \theta_2\}$ contains only one node and hence there cannot be any facet different from \widetilde{L}_1 and \widetilde{L}_2 which is hit by rays produced by the algorithm, contradicting the assumption that $E \geq 1$. Knowing that $\{\theta_1, \theta_2\}$ is divided, i.e., the root node has 2 children, consider the number of facets $e_{1,3}$ and $e_{2,3}$ of the cell contained in the cones generated by $\{\theta_1, \theta_3\}$ and $\{\theta_2, \theta_3\}$ respectively, excluding, in each cone, the facets (1 or 2) hit by the boundary rays. Since $E = e_{1,3} + e_{2,3} + 1$ (where the 1 comes from the facet hit by the ray in the direction of θ_3) it follows that $e_{1,3} < E$ and $e_{2,3} < E$. Thus the induction hypothesis implies that the trees of the restriction of the algorithm to the cones generated by the subfaces $\{\theta_1, \theta_3\}$ and $\{\theta_2, \theta_3\}$ contain exactly $2e_{1,3} + 1$ and $2e_{2,3} + 1$ nodes, respectively. Hence the tree generated by the restriction of the algorithm to the original cone (the one corresponding to $\{\theta_1, \theta_2\}$) contains $(1 + 2e_{1,3}) + (1 + 2e_{2,3}) + 1 = 2E + 1$ nodes, as claimed. \square

Lemma 9.5. *Given a cell of some site $p = P_k$, the algorithm tree contains at most $2e_k - 1$ nodes, where e_k is the number of facets of the cell.*

Proof. Consider the 3 cones generated by the very initial rays, i.e., the ones shot in the direction of the corners of the simplex. The number of nodes in the algorithm tree is the sum of nodes in each of the trees corresponding to the above cones. By Lemma 9.4 we conclude that this number is $2(e_{1,2} + e_{2,3} + e_{1,3}) + 3$, where $e_{i,j}$, $i \neq j$, $i, j \in \{1, 2, 3\}$ are the number of facets hit by intermediate rays.

Note that a facet \widetilde{L} cannot be hit by two intermediate rays belonging to two different initial cones. Indeed, assume to the contrary that this happens, say $\theta_{1,2}$ is the direction of an intermediate ray belonging to the cone generated by θ_1 and θ_2 , and $\theta_{2,3}$ is the direction of an intermediate ray belonging to the cone generated θ_2 and θ_3 . Both endpoints are located on \widetilde{L} . Then these two intermediate rays cannot be opposite, otherwise \widetilde{L} will be parallel to itself (without coinciding with itself). Hence some initial ray, in the direction of θ_2 in our notation, is contained in the (interior of the) cone generated by $\theta_{1,2}$ and $\theta_{2,3}$. As explained in the proof of Lemma 9.2, the ray of θ_2 hits \widetilde{L} at some point which must be the endpoint corresponding to this ray. This contradicts Lemma 9.2.

From the previous paragraph we can conclude that $e_{1,2} + e_{2,3} + e_{1,3} \leq e_k$. However, in fact we can conclude that $e_{1,2} + e_{2,3} + e_{1,3} \leq e_k - 2$ because the set of facets of the cell is the disjoint union of the set of facets hit by intermediate rays and the facets hit by the three initial rays, and this latter set contains at least two different facets. The assertion follows. \square

Lemma 9.6. *Consider the tree generated by the algorithm for the whole diagram. The number of nodes in this tree is $O(n)$.*

Proof. The algorithm tree for the whole diagram is the union of the trees corresponding to each cell. By Lemma 9.5 the tree of cell P_k has at most $2e_k - 1$ nodes, where e_k is the number of facets of the cell. The set of facets of each cell can be written as the disjoint union of two sets: the set of facets located on the boundary of the polygonal world X and the set of facets located on a bisecting line between P_k and another site P_j , $j \neq k$. The size w_k of the first set is not greater than the number of facets of X . Hence $\sum_{k=1}^n w_k \leq |X|n$ where $|X|$ is the number of facets of X . Denote by b_k the size of the second set. It is well known that $\sum_{k=1}^n b_k = O(n)$ (see, e.g., [6, p. 347], [40, pp. 173-5]; in [6] it is assumed that the sites are in general position; however, if this is not true, then a small perturbation of them to a general position configuration actually enlarges the number of facets as explained in [40]). Therefore $\sum_{k=1}^n e_k = \sum_{k=1}^n w_k + \sum_{k=1}^n b_k = O(n)$ and the total number of nodes in the algorithm tree of the diagram is $O(n)$. \square

Lemma 9.7. *The number of calculations needed to find the endpoint in some given direction, using Method 5.1, is bounded by a linear expression of n (but see also the final paragraph)*

Proof. As in previous lemmas, one can build the algorithm tree for Method 5.1 (see also Figure 6). Each node is a place where it is checked whether the temporary endpoint y is in the cell (using distance comparisons). If yes, then the algorithm terminates, and if not, then y is further gets closer to the given site $p = P_k$. The obtained tree is linear. Whenever a distance comparison is made with some site $a = P_j$, $j \neq k$ (or even with p itself) and it is found that $d(y, p) \leq d(y, a)$, then there is no need to consider this site in later distance comparisons since the previous inequality means that y is in the halfspace of p (with respect to a) and because y always remains on the same ray and gets closer to p , it remains in this halfspace, i.e., also later temporary endpoints y will satisfy $d(y, p) \leq d(y, a)$. However, even if $d(a, p) < d(a, y)$ then a should not be considered anymore, since this case implies that y will be moved to the intersection between its ray and the bisecting line between p and a , and so in the next iteration(s) it will satisfy $d(y, p) \leq d(y, a)$.

It follows that the total number of distance comparisons is no greater than n . A simple way to perform the above operation of not considering a given site anymore is to go over the array of sites by incrementing an index starting from the first site. By doing this each site will be accessed exactly one time and when the index will arrive to the end, the process will end. Hence the number of nodes is $O(n)$. Each calculation done in a given node is either an arithmetic operation, array manipulations, etc., i.e.,

it is $O(1)$, or it involves a some distance comparisons (each comparison is $O(1)$) and the total number of these is $O(n)$ as explained above. The assertion now follows.

As a final remark, we note that in practice, not considering a given site anymore can be applied by using more efficient methods than the one described above for reducing the number of calculations and they actually lead to $O(1)$ operations with high probability whenever the sites are generated using the uniform distribution. \square

Theorem 9.8. *The time complexity of the algorithm for the whole diagram is bounded above by an expression of the form $O(n^2)$.*

Proof. By Lemma 9.6 the number of nodes in the algorithm tree of the whole diagram is $O(n)$. The number of calculations in each node is either $O(n)$, when a new endpoint is computed (as implied by Lemma 9.7), or it is $O(1)$, when there is no need to compute a new endpoint (since the considered endpoints are already known) and only arithmetic operations, array manipulations, etc., are done. Thus at most $O(n^2)$ operations are required. \square

In the remaining part of this section we prove the correctness of the algorithm.

Lemma 9.9. *Let $F = \{\theta_1, \theta_2\}$ be a subface of the simplex and let $p + T(\theta_i, p)\theta_i$, $i = 1, 2$ be the corresponding endpoints.*

- (a) *If both endpoints are on the same facet, then the only possible vertices corresponding to F are the endpoints.*
- (b) *If one endpoint is on one facet and the other is on another one, and both facets intersect at some vertex v of the cell, then the only possible vertices corresponding to F are the endpoints and v .*

Proof. We first prove Part (a). This part seems quite obvious, but it turns out that a full proof taking into account all the details requires some work. Let $T_i = T(\theta_i, p)\theta_i$, $i = 1, 2$. By assumption, both endpoints $p + T_i$, $i = 1, 2$ are on some facet \tilde{L} of the cell. The segment $[p + T_1, p + T_2]$ is contained in \tilde{L} since \tilde{L} is convex. Suppose by way of contradiction that v is a vertex corresponding to F which is not one of the endpoints $p + T_1, p + T_2$. Then v is strictly in the cone generated by the endpoints, that is, $v = p + \lambda_1 T_1 + \lambda_2 T_2$ for some $\lambda_1, \lambda_2 \in (0, \infty)$. A simple calculation shows that the ray emanating from p in direction $v - p$ intersects the segment $[p + T_1, p + T_2]$ in exactly one point w . It must be that $w = v$, since the part of the ray beyond w is strictly outside the cell of p , the part of the ray prior to w is strictly inside the cell, and v is in the cell and it is a boundary point. As a result, $v \in [p + T_1, p + T_2] \subseteq \tilde{L}$. Since v is a vertex of the cell, it must belong to another facet \tilde{M} .

Let $v' \neq v$ be some point in \tilde{M} . Then v' cannot be on the line L on which \tilde{L} is located, since in this case $\tilde{L} \cap \tilde{M}$ will include a non-degenerate interval (the interval $[v, v']$), a contradiction to the fact that two different facets intersect at a point or do not intersect at all. In addition, v' cannot be in the half-space generated by L in which p is located. Indeed, suppose to the contrary that this happens (see Figure 9). First note that v' is not on the ray emanating from v and passing via p because this means that $[v, v']$ passes via p , hence the facet \tilde{M} passes via p , a contradiction (because p

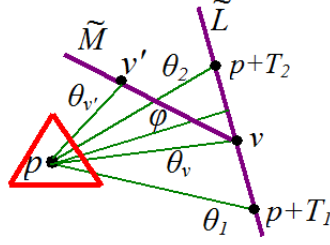


FIGURE 9. An illustration of one of the cases described in Lemma 9.9(a).

is an interior point). Consider the cone generated by the rays emanating from p and passing via v and v' . Denote $\theta_v = (v - p)/|v - p|$ and $\theta_{v'} = (v' - p)/|v' - p|$. Any ray ϕ between θ_v and $\theta_{v'}$ close enough to θ_v (i.e., its generating unit vector is close enough to θ_v) intersects the segment $[v', v) \subset \tilde{M}$, and later it intersects \tilde{L} as a simple calculation shows (using the fact that v is in the open segment $(p + T_1, p + T_2)$ and the assumption on v'). However, once a ray emanating from p intersects a facet of the cell then this point is its endpoint, namely, its remaining part beyond the point of intersection is outside the cell. Hence the point of intersection of the ray of ϕ with \tilde{L} is outside the cell, contradicting the fact that it is on \tilde{L} and hence it is in the cell.

As a result, the only possibility for v' is to be in the other half-space generated by L , a contradiction to the fact that any point in this half-space is outside the cell. This contradiction completes the proof of part (a).

Now consider Part (b). Let $\theta_3 = (v - p)/|v - p|$, and let $F_1 = \{\theta_1, \theta_3\}$ and $F_2 = \{\theta_2, \theta_3\}$. The possible vertices corresponding to F are the unions of the ones corresponding to F_1 and F_2 . Because v belongs to two different facets it follows that $p + T_i$, $i = 1, 3$ are on one facet, and $p + T_i$, $i = 2, 3$ are on another facet. By Part (a) the only possible vertices corresponding to F_1 and F_2 are their corresponding endpoints $p + T_i$, $i = 1, 2, 3$, i.e., the endpoints corresponding to F and the vertex v . \square

Lemma 9.10. *The stored points are vertices of the cell of p .*

Proof. This is evident, since each such a point u is inside the cell and it is the intersection of two edges of the cell. \square

In the following lemmas we use the concept of a “prime subface”, namely a subface created by the algorithm at some stage but which has not been further divided after its creation. By Lemma 9.3 there are finitely many prime subfaces. Any two such subfaces either do not intersect or intersect at exactly one point (their corner), and their union is the simplex around p . See Figure 10 for an illustration.

Lemma 9.11. *Let v be a vertex of the cell and assume that it coincides with an endpoint corresponding to a corner of a prime subface F . Then v , as a vertex, is found by the algorithm.*

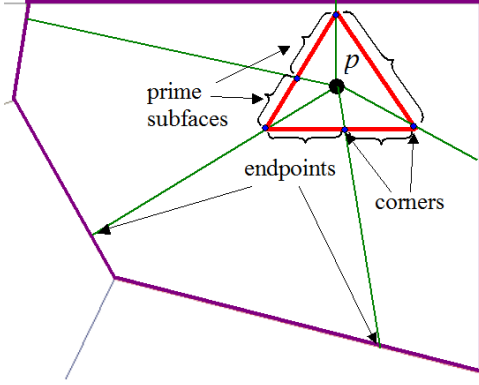


FIGURE 10. Prime subfaces and their rays.

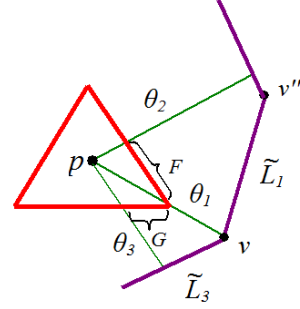


FIGURE 11. Illustration of Lemma 9.11.

Proof. The proof is not immediate as it may perhaps seem at first, since a vertex is found by the algorithm only after an intersection between two facets of the cell is detected, and so although v was found, as an endpoint, it is still not known that it is a vertex, and further calculations are needed in order to classify it as a vertex.

Assume by way of contradiction that v , as a vertex, is not found. Let $p + T_1$ and $p + T_2$ be the endpoints corresponding to the corners of F . Let \tilde{L}_1, \tilde{L}_2 be the corresponding facets of the cell on which these endpoints are located. The facets $\tilde{L}_i, i = 1, 2$ are located on corresponding lines L_1, L_2 . By assumption v coincides with one of the endpoints, say with $p + T_1$.

Since $p + T_1$ corresponds to a corner of F , this corner is located on another prime subface G . Therefore v is also in the cone corresponding to G . Let $p + T_3 = p + T(\theta_3, p)\theta_3$ be the other endpoint corresponding to G . It is located on some facet $\tilde{L}_3 \neq \tilde{L}_1$ of the cell. Consider the lines L_1 and L_3 corresponding to \tilde{L}_1 and \tilde{L}_3 respectively. If they do not intersect, or intersect at a point outside the cone corresponding to G or inside the cone but outside the cell, then by the definition of the algorithm G must be divided, a contradiction (G is assumed to be prime). Since $L_1 \neq L_3$, it follows that L_1 and L_3 intersect at a point v_{13} which is in the cone and in the cell, i.e., a vertex.

Any facet of the cell intersects exactly two additional facets. In particular this is true for \tilde{L}_1 : one intersections occurs at the vertex v and another one at another vertex v'' located on \tilde{L}_1 in the direction (along \tilde{L}_1) from v to $p + T_2$. Consider the line passing via p and v : it separates the plane into two halfplanes. One of them contains the ray θ_2 and actually \tilde{L}_1 (and hence also v''). The other contains the ray θ_3 (and intersects \tilde{L}_1 only at v) and hence also the facet \tilde{L}_3 . Thus \tilde{L}_3 cannot intersect \tilde{L}_1 at the halfplane of v'' . Since we know that \tilde{L}_3 intersects \tilde{L}_1 this can only be at v . See Figure 11. Hence $v_{13} = v$ but since v_{13} is a vertex, this means that v is found as a vertex when considering the subface G during the running time of the algorithm (line 21), a contradiction. \square

Lemma 9.12. *The algorithm finds all the vertices and edges of the cell.*

Proof. By Lemma 9.3 the algorithm terminates after a finite number of steps. Let $(F_j)_{j=1}^s$ be the finite list of all prime subfaces. Assume by way of contradiction that some vertex u is not found. Then u corresponds to some point located on some prime subface F of the simplex, since the ray in direction $u - p$ intersects the simplex at exactly one point, and each point on the simplex belongs to some prime subface.

Let $p + T_1$ and $p + T_2$ be the endpoints corresponding to the corners of F , and let $\widetilde{L}_1, \widetilde{L}_2$ be the corresponding facets of the cell on which these endpoints are located. The facets $\widetilde{L}_i, i = 1, 2$ are located on corresponding lines L_1, L_2 . Let B the matrix from (3.1).

Assume first that $\det(B) = 0$. Then it must be that $L_1 = L_2$, since otherwise L_1 and L_2 are parallel and hence F is divided into two subfaces ((line 14), a contradiction. However, if $L_1 = L_2$, then $\widetilde{L}_1 = \widetilde{L}_2$, and hence, by Lemma 9.9(a), the only possible vertices corresponding to F are the endpoints $p + T_1, p + T_2$. In particular, the missing vertex u coincides with one of these endpoints. But then, according to Lemma 9.11, the algorithm finds u as a vertex, a contradiction.

Assume now the case $\det(B) \neq 0$. Then $L_1 \neq L_2$. It must be that λ from (3.1) is nonnegative, since otherwise F is divided into two subfaces (line 26), a contradiction. The point of intersection between L_1 and L_2 is $v = p + \lambda_1 T_1 + \lambda_2 T_2$, and it must be in the cone corresponding to F and also in the cell of p , since otherwise F is divided into two subfaces (line 23). Hence v is a vertex corresponding to F and it is found by the algorithm at a stage when (3.1) is considered. If $v = u$, then the algorithm finds u when treating F , a contradiction. Hence $v \neq u$, and by Lemma 9.9(b) it must be that u coincides with one of the endpoints $p + T_1$ or $p + T_2$. But then, according to Lemma 9.11, the algorithm finds u as a vertex, a contradiction.

Therefore all the vertices of the cell are detected. As explained in Method 5.1 and Section 6, when a vertex is detected, also the edges which intersect at it are detected. Since all the possible vertices are found by the algorithm, then so are all the possible edges. \square

Proof of Theorem 8.1. This is a simple consequence of Theorem 9.8 and Lemmas 9.3, 9.10, and 9.12. \square

ACKNOWLEDGMENT

I am indebted to Omri Azencot for implementing the algorithm so expertly and for helpful discussions. I also thank Renjie Chen for helpful discussion regarding [16].

REFERENCES

1. A. Aggarwal, B. Chazelle, L. J. Guibas, C. O'Dúnlaing, and C. K. Yap, *Parallel computational geometry*, Algorithmica **3** (1988), 293–327, preliminary version in FOCS 1985, pp. 468–477.
2. A. Aggarwal, L. J. Guibas, J. Saxe, and P. W. Shor, *A linear-time algorithm for computing the Voronoi diagram of a convex polygon*, Discrete Comput. Geom. **4** (1989), no. 6, 591–604, A preliminary version in STOC 1987, pp. 39–45.
3. O. Aichholzer, W. Aigner, F. Aurenhammer, T. Hackl, B. Jüttler, E. Pilgerstorfer, and M. Rabl, *Divide-and-conquer for Voronoi diagrams revisited*, Proceedings of the 25th annual ACM Symposium on Computational Geometry (SoCG 2009), 2009, pp. 189–197.

4. N. M. Amato, M. T. Goodrich, and E. A. Ramos, *Parallel algorithms for higher-dimensional convex hulls*, Proceedings of the 35th IEEE Symposium on Foundations of Computer Science (FOCS 1994), 683–694.
5. N. M. Amato and F. P. Preparata, *The parallel 3D convex hull problem revisited*, Internat. J. Comput. Geom. Appl. **2** (1992), 163–173.
6. F. Aurenhammer, *Voronoi diagrams - a survey of a fundamental geometric data structure*, ACM Computing Surveys, vol. 3, 1991, pp. 345–405.
7. F. Aurenhammer and R. Klein, *Voronoi diagrams*, Handbook of computational geometry (J. Sack and G. Urrutia, Eds.) (2000), 201–290.
8. C. B. Barber, D.P. Dobkin, and H.T. Huhdanpaa, *The Quickhull algorithm for convex hulls*, ACM Transactions on Mathematical Software **22** (1996), 469–483.
9. C. B. Barber and The-Geometry-Center, *Qhull (software)*, (2011), Copyright: 1993-2011. Web: <http://www.qhull.org>.
10. J. L. Bentley, B. W. Weide, and A. C. Yao, *Optimal expected-time algorithms for closest point problems*, ACM Trans. Math. Softw. **6** (1980), 563–580.
11. G. E. Blelloch, J. C. Hardwick, G. L. Miller, and D. Talmor, *Design and implementation of a practical parallel Delaunay algorithm*, Algorithmica **24** (1999), 243–269.
12. J. F. Bonnans, J. C. Gilbert, C. Lemarechal, and C. A. Sagastizábal, *Numerical Optimization: Theoretical and Practical Aspects*, 2nd ed., Universitext, Springer-Verlag, Berlin, 2006.
13. K. Q. Brown, *Voronoi diagrams from convex hulls*, Inf. Process. Lett. **9** (1979), 223–228.
14. ———, *Geometric transforms for fast geometric algorithms*, Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, 1980.
15. T. M. Chan and E. Y. Chen, *Optimal in-place and cache-oblivious algorithms for 3-d convex hulls and 2-d segment intersection*, Computational Geometry **43** (2010), 636–646, Special issue on SoCG 2009 (a preliminary version: SoCG 2009 pp. 80–89).
16. R. Chen and C. Gotsman, *Localizing the Delaunay triangulation and its parallel implementation*, Proceedings of the 9th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2012), 2012, Rutgers University, NJ, USA, pp. 24–31.
17. A. Chow, *Parallel algorithms for geometric problems*, Ph.D. thesis, University of Illinois, Urbana, 1980.
18. R. Cole, M. T. Goodrich, and C. O’Dúnlaing, *A nearly optimal deterministic parallel Voronoi diagram algorithm*, Algorithmica **16** (1996), 569–617.
19. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, second ed., MIT Press and McGraw-Hill, 2001.
20. N. Dadoun and D. G. Kirkpatrick, *Parallel construction of subdivision hierarchies*, J. Comput. Syst. Sci. **39** (1989), 153–165.
21. G. B. Dantzig, *Linear programming and extensions*, Princeton University Press, Princeton, N.J., 1963.
22. M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational geometry: Algorithms and applications*, third ed., Springer, 2008.
23. Q. Du, M. Emelianenko, and L. Ju, *Convergence of the Lloyd algorithm for computing centroidal Voronoi tessellations*, SIAM J. Numer. Anal. **44** (2006), 102–119.
24. Q. Du, V. Faber, and M. Gunzburger, *Centroidal Voronoi tessellations: applications and algorithms*, SIAM Rev. **41** (1999), no. 4, 637–676.
25. D. J. Evans and I. Stojmenović, *On parallel computation of Voronoi diagrams*, Parallel Computing **12** (1989), 121–125.
26. S. Fortune, *A sweepline algorithm for Voronoi diagrams*, Algorithmica **2** (1987), 153–174, A preliminary version in SoCG 1986, pp. 313–322.
27. C. Gold, *The Voronoi Web Site*, 2008, http://www.voronoi.com/wiki/index.php?title=Main_Page.

28. M. T. Goodrich, C. O'Dúnlaing, and C. Yap, *Computing the Voronoi diagram of a set of line segments in parallel*, *Algorithmica* **9** (1993), 128–141, preliminary version in LNCS WADS 1989, pp. 12–23.
29. P. J. Green and R. Sibson, *Computing Dirichlet tessellations in the plane*, *Comput. J.* **21** (1977), 168–173.
30. L. Guibas, D. Knuth, and M. Sharir, *Randomized incremental construction of Delaunay and Voronoi diagrams*, *Algorithmica* **7** (1992), 381–413.
31. L. J. Guibas and J. Stolfi, *Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams*, *ACM Trans. Graph.* **4** (1985), 74–123.
32. C. A. R. Hoare, *Quicksort*, *The Computer Journal* **5** (1962), 10–16.
33. G. J. Hwang, J. M. Arul, E. Lin, and C.-Y. Hung, *Design and multithreading implementation of the wave-front algorithm for constructing Voronoi diagrams*, *Distributed and Parallel Computing*, vol. 3719, 2005, pp. 257–266.
34. V. Klee, *On the complexity of d -dimensional Voronoi diagrams*, *Arch. Math.* **34** (1980), 75–80.
35. V. Klee and G. J. Minty, *How good is the simplex algorithm?*, *Inequalities III* (Proceedings of the Third Symposium on Inequalities held at the University of California, Los Angeles, Calif., September 19, 1969, dedicated to the memory of Theodore S. Motzkin) (O. Shisha, ed.), Academic Press, New York-London, 1972, pp. 159–175.
36. F. Lee and R. Jou, *Efficient parallel geometric algorithms on a mesh of trees*, *Proceedings of the 33rd annual ACM Southeast Regional Conference*, 1995, pp. 213–218.
37. H. Meyerhenke, *Constructing higher-order Voronoi diagrams in parallel*, *EWCG 2005*, pp. 123–126.
38. T. Ohya, M. Iri, and K. Murota, *Improvements of the incremental methods for the Voronoi diagram with computational comparison of various algorithms*, *J. Operations Res. Soc. Japan* **27** (1984), 306–337.
39. A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu, *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*, second ed., *Wiley Series in Probability and Statistics*, John Wiley & Sons Ltd., Chichester, 2000, with a foreword by D. G. Kendall.
40. J. O'Rourke, *Computational geometry in C*, Cambridge University Press, New York, 1994.
41. S. Rajasekaran and S. Ramaswami, *Optimal parallel randomized algorithms for the Voronoi diagram of line segments in the plane and related problems*, *Proceedings of the 10th annual ACM Symposium on Computational Geometry (SoCG 1994)*, pp. 57–66.
42. D. Reem, *An algorithm for computing Voronoi diagrams of general generators in general normed spaces*, *Proceedings of the sixth International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2009)*, pp. 144–152.
43. J. H. Reif and S. Sen, *Optimal parallel randomized algorithms for three dimensional convex hulls and related problems*, *SIAM J. Comput.* **21** (1992), 466–485.
44. O. Schwarzkopf, *Parallel computation of discrete Voronoi diagrams*, *LNCS* **349** (1989), 193–204, (Proc. of STACS 1989).
45. O. Setter, M. Sharir, and D. Halperin, *Constructing two-dimensional Voronoi diagrams via divide-and-conquer of envelopes in space*, *Transactions on Computational Science* **IX** (2010), 1–27, a preliminary version in ISVD 2009, pp. 43–52.
46. M. I. Shamos and D. Hoey, *Closet-point problems*, *Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1975)*, pp. 151–162.
47. M. Sharir and P. Agarwal, *Davenport-Schinzel sequences and their geometric applications*, Cambridge University Press, 1995.
48. J. R. Shewchuk, *Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator*, *Applied Computational Geometry: Towards Geometric Engineering* (Ming C. Lin and Dinesh Manocha, eds.), *Lecture Notes in Computer Science*, vol. 1148, Springer-Verlag, 1996, From the First ACM Workshop on Applied Computational Geometry, pp. 203–222.

- 49. ———, *Delaunay refinement algorithms for triangular mesh generation*, Comput. Geom. **22** (2002), 21–74.
- 50. D. A. Spielman, S.-H. Teng, and A. Üngör, *Parallel Delaunay refinement: Algorithms and analyses*, International Journal of Computational Geometry and Applications **17** (2007), 1–30.
- 51. C. Trefftz and J. Szakas, *Parallel algorithms to find the Voronoi diagram and the order- k Voronoi diagram*, Proc. of the International Parallel and Distributed Processing Symposium (IPDPS 2003).
- 52. B. C. Vemuri, R. Varadarajan, and N. Mayya, *An efficient expected time parallel algorithm for Voronoi construction*, In Proceedings of SPAA 1992, pp. 392–401.